Journal of Statistical Software

MMMMMM YYYY, Volume VV, Issue II. *http://www.jstatsoft.org/*

JavaScript Object Notation (JSON) and R
The RJSONIO package

Duncan Temple Lang
University of California at Davis

Abstract

The JavaScript Object Notation (JSON) format is becoming widely used as a means of exchanging data. It is used in Web Services, client server applications and to some extent as a means of serializing data for use between applications. We describe an R package that facilitates both importing data from JSON into R and also exporting R objects as JSON content. The package is quite simple but the architecture allows R programmers to customize several aspects of the computations.

Keywords: Input/Output, application independent data format, JavaScript, Web Services.

1. Things To Fix

1. Character Encoding
2. Base64 Encoding

2. JavaScript Object Notation

Purpose of JSON and the need to be able to deal with it in R. Common uses. REST, inserting objects in to JavaScript code e.g. HTML documents. ECMAScript and ActionScript for Flash.

Comparison with XML. Not a competition, just different purposes and strengths.

The **RJSONIO** package offers two primary operations - transforming JSON content into R objects and serializing R objects to JSON format. This allows us to import and export JSON from R. The two functions that do this are `fromJSON()` and `toJSON()`. `fromJSON()` can read JSON content from a file or a general connection, or from a string in memory. The latter is convenient when we obtain the JSON content from some other computation such as a Web request and the content is already in memory.

Character encoding

Before developing the **RJSONIO** package, we used **rjson** (Couture-Beil 2011-6-26) when serializing R objects into JavaScript code within HTML pages. This worked very well for small objects but was too slow for large objects. **RJSONIO** solves some of the speed issues, primarily by vectorizing the code that generates the content. We developed the **RJSONIO** package to be a direct substitute for **rjson** so code that used the latter would not need to be changed to use **RJSONIO**. So we can think of **RJSONIO** as a second-generation of **rjson** with a focus on efficiency which was not warranted when **rjson** was being developed as means to effect entirely new facilities for **R**.

RJSONIO also changes the approach used to parse JSON content into R. Firstly, it uses a C library - *libjson* (Wallace 2011). This should yield two benefits. Firstly, it should be faster than pure interepreted R code. Secondly, it relies on code that is used developed by others and used in other applications. The benefit of this is that we do not have to maintain it and we benefit from any updates as they are made in the *libjson* project. Relying on *libjson* means that we also suffer from its deficiencies and bugs and do not have the flexibility to design things as we want. The hope is that **libjson** is used in other projects so that bugs will be identified by a larger audience than if we had developed the code ourselves for use only in R. Unfortunately, this may not be the case. Since we use **libjson**, it would appear we have an additional dependency which users must satisfy. However, to simplify installation for users, we have included a copy of the *libjson* code. We use that version only if we cannot find a version of **libjson** on the local machine to which we are installing the source package. This means that R users can elect to use newer versions of **libjson** but do not have to.

While **RJSONIO** acts as a direct replacement for **rjson**, it also offers additional features and controls.

The remainder of this paper is organized as follows. We start with a brief description of the simple but general JSON format. In Section 4, we illustrate how to read JSON content into R in three different contexts: read local JSON files from Kiva.org, parsing results from a request to the Twitter API, and interacting with CouchDB, a simple client-server database. We then discuss how to serialize R objects in JSON format. In section Section 6, we discuss how one can customize the parsing of JSON content. We end with some notes about how the package could be extended and made more general.

3. The JSON Format

The JSON format is quite simple and reflects the basic data structures in JavaScript and other programming languages. We'll start with scalar values. Logical values are represented by `true` and `false`. There is no distinction in JSON between an integer and a real valued number. Scientific notation, e.g., 123e10 and 123E-5 is supported. Strings are enclosed within pairs of ", i.e. the double quote character.

Arrays are ordered collections of values. Each element is separated by a comma (,). Associative arrays have names for each of the elements. These are equivalent to named lists in R although the order is not guaranteed. Regular arrays are enclosed by `[ ]` pairs, again with elements being separated by a comma. Associative arrays are enclosed within `{ }`. Each element is given in the form `name: value`. The `name` term should be enclosed within quotes. Not all JSON parsers insist on this (including **libjson** and hence **RJSONIO**), but it is good practice to ensure these names are quoted.

Each element in an associative or regular array can be an arbitrary JSON object. This allows us to nest values so that we can have an array whose elements are arrays, associative arrays and scalar values.

The final element of the format is the literal value `null`. It represent the null object in **JavaScript** and is a special constant object there, useful for comparing the value of a variable to this special state. In some senses, it corresponds to `NULL` in **R**. However, it might also map to an empty vector.

The format is very simple Note that it does not have support for mathematical terms such as infinity, pi, e. Nor does it have the notion of a `NA`, the missing value in **R**.

Valid JSON requires that the top-level content be either an array or an object. This means that simple literal values such as "2" or 'abc' are not valid, but `"[2]"` and `"{xyz: 'abc'}"` are valid.

How do we map `null` to a value in **R**? How do we map empty vectors in **R** to JSON?

JSON is written as plain text. It would appear that we cannot include binary content such as an image. There is however a way around this. We can take arbitrary binary content and convert it to text using base-64 encoding commonly used to include binary content in email messages. There are several implementation of functions that convert to and from base64 encoding in various R packages include **caTools**, **RCurl** and **readMzXmlData**.

While we can easily include binary content in JSON using base64 encoding, it is imperative that the consumer of that JSON content be aware that the content is base64 and so can decode it appropriately. Unfortunately, JSON doesn't provide a standard or convenient mechanism for identifying meta-data about elements of the content.

Valid JavaScript and can be evaluated. Security concerns.

4. Examples

4.1. Reading Non-Rectangular Data

While many data sets come to us as rectangular tables made up of rows and columns corresponding to observations each with the same number of variables. This works reasonably well, but is not rich enough for many more complex data structures.

We may have repeated measurements for different observational units and so not the same number of variables in each "row". For each observation, we might have hierarchical structures such as their address or location. We could collapse this into separate variables at the top-level, but this might be a different format for different types of observational units. So in short, we need a richer format to represent raw data before we project it into a rectangular format or data frame in **R**

An example of a moderately complex data set is the dump of the Kiva database from Kiva.org. Kiva is an non-profit organization that connects lenders and borrowers on-line to provide micro-loans for people in developing countries. They make several details of loans, borrowers and lenders available both via a Web Service API and also via serializing their database. The provide this serialization in both XML and JSON formats. The data can be downloaded from `http://build.kiva.org/`, specifically `http://s3.kiva.org/snapshots/kiva_ds_json.zip`. We download and extract the files and this produces two directories, one for lenders and another for loans. Each of these contains a collection of files with the .json extension each numbered from 1 to the number of files in that directory.

We'll look at the loan files. We can read one of these files with

```
loans1 = fromJSON("loans/1.json")
```

The result in **loans1** is a list with two elements. The first is named "header" and provides information about the contents of the file, e.g. the number of loans, the date it was serialized. The second element ("loans") contains the data for each loan. Strangely, there are 795 repeated elements which we can identify by examining the "id" element. This has nothing to do with JSON but the way the data were dumped from the database. The same occurs in the XML version. So we remove the duplicates with:

```
w = duplicated(sapply(loans1$loans, `[[`, "id"))
loans1$loans = loans1$loans[ ! w ]
```

Now we can look at each loan. We can look at the types of each element:

```
table(unlist(lapply(loans1$loans, function(x) sapply(x, class))))

character       list   logical    numeric
     8003       5000        22       5963
```

So each element has 5 lists, e.g. description, terms, location, borrowers. The location is made up of several fields identifying the town and country and also latitude and longitude in a separate list named "geo":

```
loans1$loans[[1]]$location

$country_code
[1] "UG"

$country
[1] "Uganda"

$town
[1] "Tororo"

$geo
$geo$level
[1] "town"

$geo$pairs
[1] "0.75 34.083333"

$geo$type
[1] "point"
```

How we chose to represent and work with this data in **R** depends on what we want to do with it.

The lenders data can be read in the same manner. It has a simpler structure with all but one variable for each lender being a simple scalar. Not all lenders have all variables so we have a ragged array again. However, we could easily put this data into rectangular form by having `NA` values.

Comparison with XML and speed for overall processing or XPath to get sub-elements. XQuery also.

4.2. Web Services

JSON is commonly used in Web Services, specifically as the result format in REST (Representational State Transfer) services. The idea is that we make an HTTP request to query information we want. We specify a URL and possibly additional arguments to parameterize our request. Let's use the Twitter API as an example. Twitter allows us to query the the 20 most recent public "statues" or activities on Twitter. We send a request to the URL `http://api.twitter.com/1/statuses/public_timeline`. We can control the format of the result by appending one of the strings "xml", "json", "rss" or "atom", separated by a period.

```
url = "http://api.twitter.com/1/statuses/public_timeline"
txt = getURLContent(sprintf("%s.json", url))
```

This returns a string containing the JSON content. This object also has attributes that identify the content type ("application/json") and the character encoding. These are extracted from the header of the HTTP response. [1]

Now that we have the JSON content as a string, we can convert it to R values via `fromJSON()`. We do this with

```
tweets = fromJSON(tt, asText = TRUE)
```

We use the **asText** argument to ensure the function does not confuse the value as the name of a file. The function will typically guess correctly, but since we know we have the JSON content as a string, it is good practice to indicate this to `fromJSON()`.

The result is an **R** list with twenty elements. Each element is also a list with 19 named elements:

```
names(tweets[[1]])
```

```
 [1] "place"                      "in_reply_to_user_id"
 [3] "user"                       "in_reply_to_status_id"
 [5] "text"                       "id_str"
 [7] "favorited"                  "created_at"
 [9] "in_reply_to_status_id_str" "geo"
[11] "in_reply_to_screen_name"    "id"
[13] "in_reply_to_user_id_str"    "source"
[15] "contributors"               "coordinates"
[17] "retweeted"                  "retweet_count"
[19] "truncated"
```

The "user" element is also a list:

```
names(tweets[[1]]$user)
```

[1] In older versions of **RCurl**, this was returned as a binary object. Now, **RCurl** recognizes the content type "application/json" as text.

```
 [1] "statuses_count"
 [2] "notifications"
 [3] "profile_text_color"
 [4] "protected"
 [5] "default_profile"
 [6] "profile_sidebar_fill_color"
 [7] "location"
 [8] "name"
 [9] "profile_background_tile"
[10] "listed_count"
[11] "contributors_enabled"
[12] "profile_background_image_url_https"
[13] "utc_offset"
[14] "url"
[15] "id_str"
[16] "following"
[17] "verified"
[18] "favourites_count"
[19] "profile_link_color"
[20] "profile_image_url_https"
[21] "description"
[22] "created_at"
[23] "profile_sidebar_border_color"
[24] "time_zone"
[25] "profile_image_url"
[26] "is_translator"
[27] "default_profile_image"
[28] "profile_use_background_image"
[29] "id"
[30] "show_all_inline_media"
[31] "geo_enabled"
[32] "friends_count"
[33] "profile_background_color"
[34] "followers_count"
[35] "screen_name"
[36] "profile_background_image_url"
[37] "follow_request_sent"
[38] "lang"
```

Compelling example: NYTimes? What others? Gloss over the RCurl requests.

4.3. CouchDB

Mention that others have built an R-CouchDB interface.

5. Creating JSON Content from R

To this point, we have seen how we can consume or import JSON content in **R**. We now turn our attention to how we create JSON content from **R** and so export it to other applications. Basically we want to generate text that we store as a string or write to a connection and which consists of JSON content. Any R programmer can create arbitrary JSON content using R commands such as `paste()`, `sprintf()` and `cat()` and character vectors or connections (including `textConnection()`). We focus here however on serializing arbitrary R objects in JSON format so that the information can be restored within another JSON-enabled application. The basic function that takes an R object and serializes it to a JSON string is `toJSON()`. This function takes an R object and serializes its elements. Basically, this maps R vectors (logical, integer, numeric, character) to either a dictionary (or object in JSON terms) or a regular array. If the R vector has a names, we preserve these and use a dictionary.

```
x = c(a = 1, b = 10, c = 20)
toJSON(x)
```

There are occasions when we have names on an R object, but we want the resulting JSON value to be a simple array. We can use the **.withNames** parameter to control this. Passing a value of `FALSE` causes the names to be ignored and a regular array to be created, e.g.

```
x = c(a = 1, b = 10, c = 20)
toJSON(x, .withNames = FALSE)
```

There are methods for serializing R objects to JSON for various classes of R objects. These allow us to customize how some R objects are translated to JSON. For example, a matrix in R is merely a one-dimensional vector with a dim attribute that allows R to treat as a two or more dimensional object. As a result, by default, it would be serialized as a single long vector in column-wise order. However, a matrix might be represented in **JavaScript** as an array of row arrays, i.e. a top-level container in which element is itself a one-dimensional array for a given row. So we define a method to handle *matrix* objects in R. It is defined as

```
setMethod("toJSON", "matrix",
    function(x, container = length(x) > 1 ||
                                 length(names(x)) > 0,
             collapse = "\n", ..., .level = 1L,
             .withNames = length(x) > 0 && length(names(x)) > 0) {
      tmp = paste(apply(x, 1, toJSON),
                  collapse = sprintf(",%s", collapse))
      if(!container)
        return(tmp)

       if(.withNames)
         paste("{", paste(dQuote(names(x)), tmp, sep = ": "), "}")
       else
         paste("[", tmp, "]")
    })
```

With this defined, the code

```
toJSON(matrix(1:10, 5, 2))
```

yields

```
[ [ 1, 6 ],
  [ 2, 7 ],
  [ 3, 8 ],
  [ 4, 9 ],
  [ 5, 10 ] ]
```

`toJSON()` and its methods could be extended to write to a connection. The default connection could be a `textConnection()` and if this was not specified (i.e. missing in the initial call), the string rather than the connection would be returned. This would allow us to avoid collecting the JSON text in memory for an entire object and to emit/flush content to a connection as it was generated. This would save memory and could be important for large objects.

6. Customizing the Parser

We can provide our own handlers to process each element as it is encountered by the JSON parser. This is similar to the SAX style of parsing for XML.

7. Future Directions

At present, we omit/drop attributes on R objects when serializing to JSON. We use the length, dim and names for vectors, but ignore them for other types of R objects and ignore any other attributes entirely. To serialize to R Attributes on R objects. We could use either an empty name or .Data for the data part of an object and then "attributes" to identify the list of attributes.

We cannot serialize R functions easily to JavaScript as they do not make a great deal of sense in that language. Instead, we can serialize the source code for a function as a string. This loses information, and this is very important if the function has a non-standard environment. We have also experimented with approaches to translating the R syntax to JavaScript code and possibly R code to JavaScript in an effort to simplify authoring JavaScript code for use in, e.g., Web pages.

The **libjson** parser expects the entire JSON content to be in memory when it starts to read, i.e. passed as a single string. We would like to be able to have the parser read from a file or connection and access additional bytes of the input stream as it requires them. This would reduce the memory required as we wouldn't have to load the file into memory ahead of time. Instead, we would retain a smaller buffer of content that is being processed.

We have developed a bi-directional interface between the JavaScript interpreter SpiderMonkey used in the Mozilla/Firefox browser. This allows us to pass R objects to JavaScript and vice verse using C-level references. However, we can also transfer objects by value between the two languages using **RJSONIO** with very little infrastructure.

References

Couture-Beil A (2011-6-26). "The rjson package."

Wallace J (2011). "libjson." URL `http://libjson.sourceforge.net`.

Affiliation:

Firstname Lastname
Affiliation
Address, Country
E-mail: `name@address`
URL: `http://link/to/webpage/`

Journal of Statistical Software `http://www.jstatsoft.org/`
published by the American Statistical Association `http://www.amstat.org/`
Volume VV, Issue II *Submitted:* yyyy-mm-dd
MMMMMM YYYY *Accepted:* yyyy-mm-dd